실험 3. 조금 복잡한 입출력 실험-3

1. 목적

많은 임베디드 시스템들은 주어진 입력에 따라 어떤 동작을 하는데, 그의 구현을 위하여 시스템 내부에 상태를 유지해야 한다. 따라서, 많은 경우 소프트웨어 구현에 유한 상태 기계(Finite State Machine)을 기반으로 알고리즘을 구현한다. 이 실험에서는 State Machine을 구현하는 방법과 고려해야 사항을 배우는 것을 목표로 한다.

또 많은 응용에서 '대기' 상태가 발생하는 바, 실시간 운영체제가 제공하는 타이머를 이용하여 대기 상태를 구현하고, 또 시간 정보를 소프트웨어 구현에 이용하는 방법을 배운다.

2. 예제 내용

이 예제는 스위치 A (KEY_A)를 짧게 Click을 하면 BARLED1이 모두 켜지고 BARLED2는 꺼지며, 길게 Click 하면 (즉, 오래 눌렀다 떼면) 반대로 BARLED2가 모두 켜지고 BARLED1는 꺼지는 프로그램이다. 즉, 스위치가 눌렸다가 떨어지면, 눌려있던 시간을 기준으로 해당 LED를 켠다. LED를 켜고 끄는 시점은, 스위치를 누르고 있던 시간의 측정이 필요하므로, 당연히 스위치가 손에서 떨어질 때이다.

●(LED 켜짐) ○ (LED 꺼짐)

입력	BARLED1	BARLED2
After Short Click	●	○
After Long Click	○	●
초기 상태	○	○

표 1. 문제의 정의

위에서 Short Click은 300msec 이하 시간 동안 누른 것을, Long Click은 그 이상 길이로 누른 것으로 정의한다.

이 동작은 간단하기는 하지만, 이런 류의 동작에는 스위치, LED 등 여러 개의 상태의 유지가 필요하며, 이를 위해 소프트웨어가 유지해야하는 상태표(또는 상태 다이어그램)를 만들고 그에 따라 프로그램을 구현하면 시스템의 로직이 상태표에 의해 정의되기 때문에, 디버깅의 대상이 코드가 아니라, 상태표가 되어, 설계, 구현, 시험이 모두 쉬워진다. State Machine을 기반으로 어떤 시스템을 구현하는 방법은 상태와 입출력의 정의를 어떻게 하는가에 따라, 매우 다양하게 존재할 수 있다.

상태표에 의한 구현 방법

상태표는 상태를 정의하고 입력 이벤트에 따라 어떻게 상태가 전이되며, 전이되는 순간 어떤 동작을 하는가를 정의한 표이다. 즉, 상태표는 현 상태와 입력, 그 입력에 따른 다음 상태와 출력(동작)으로 정의된다. 길게 누르기, 짧게 누르기 두 가지를 관리하기 위하여, 이 방법에서 유지해야하는 상태는 다음과 같이 정의된다.

- 상태 0: 초기 상태 (프로그램이 처음 시작된 상태, 스위치가 눌려있지 않은 상태)
- 상태 1: 스위치가 눌린 뒤, 아직 200msec가 안되어, 계속 시간을 재고 있는 상태
- 상태 2: 스위치가 눌린 뒤 200msec가 지난 채 스위치가 눌려있는 상태

이 상태 머신의 입력은 다음 세 가지로 스위치 조작, Timeout 등이다.

- SW-ON : 스위치가 ON
- SW-OFF : 스위치가 OFF
- TO (Time Out) : 타이머 시작 후 200msec가 지남

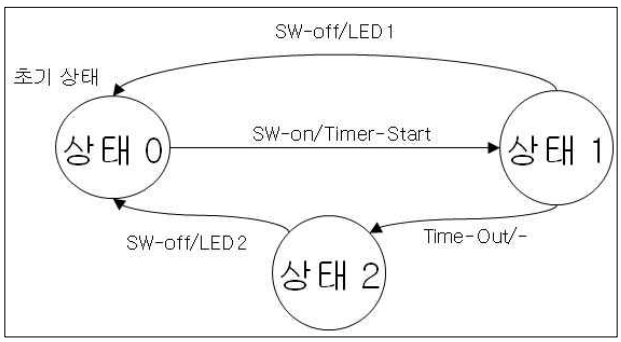
상태의 전이에 따른 동작은 다음 세 가지로 상태 전이 때 수행된다.

- LED1 : BARLED1을 켜고 BARLED2를 끄는 동작
- LED2 : 반대로 BARLED1을 끄고, BARLED2를 켜는 동작
- TS (Timer Start) : 스위치를 누르는 순간, 시간 재기를 초기화하는 동작

< 상태표와 상태 다이어그램 >

상태번호	다음 상태/동작		
	SW-ON	SW-OFF	TO
0	1/TS	-/-	-/-
1	1/-	0/LED1	2/-
2	2/-	0/LED2	-/-

('-'는 현상태 유지 / 무동작을 의미)



3. 예제 소스

예제 소스의 함수 Exp_3_Sample_A()는 switch() 문을 이용하여 위 상태표의 세 가지 상태에 대하여 각 상태에 해당하는 입력을 if 문으로 확인하여 처리하는 방식의 소스이다. 이 방법은 상태와 입력, 입력에 따른 동작이 비교적 간단할 때 사용할 수 있는 방법이다.

```
while (1) {
  SWITCH_ON = (NDS_SWITCH() & KEY_A);
  switch (state) {
    case 0 :
      if (SWITCH_ON) {
        state = 1;                          // Next State
        start_time = xTaskGetTickCount();   // Action
      }
      break;
    case 1 :
      if (!SWITCH_ON) {
         state = 0;                         // Next State
         writeb_virtual_io(BARLED1, 0xFF); // Action
         writeb_virtual_io(BARLED2, 0);
      } else if ((xTaskGetTickCount() - start_time) >= MSEC2TICK(300)) {
         state = 2;                         // Timeout !
      }
      break;
    case ...
```

예제 소스의 함수 Exp_3_Sample_B()는 같은 상태표를 상태 구조체 배열 방식으로 구현한 방법이다. 상태 배열(Exam_SM_B)는 각 상태 별로 입력에 따라, 다음 상태와 다음 상태로 전이하는 과정에서의 동작을 정의한 2차원 배열로, 문제에서 정의된 상태표를 그대로 옮긴 것이다.

```
#define NUM_STATE    3
#define NUM_INPUT    3

struct state_machine_b {
  int next_state
  int action
};

enum { SW_ON, SW_OFF, TO };
enum { NOP, LED1, LED2, TS };

struct state_machine_b Exam_SM_B[NUM_STATE][NUM_INPUT] = {
    //  SW_ON       SW_OFF        TO
    { { 1, TS  }, { 0, NOP  }, { 0, NOP } },    /* State 0 */
    { { 1, NOP }, { 0, LED1 }, { 2, NOP } },    /* State 1 */
    { { 2, NOP }, { 0, LED2 }, { 2, NOP } }     /* State 2 */
};
```

실제 구동 부분 프로그램은 다음과 같이 Event를 생성하는 부분 (Step 0), 동작을 구현하는 부분 (Step 1), 다음 상태를 결정하는 부분 (Step 2) 세단계로 이루어져 있다. Exp_3_Sample_B()의 방법은 상태와 입력이 많은 경우에도 소스가 언제나 같은 형태로 만들어지는 장점이 있으며, 입력(event)이 생성되는 과정과, 상태표를 정의한 구조체만 디버깅하면 되므로 구현, 시험, 수정이 매우 용이하다.

```
while (1) {

/* Step 0: Generate Input Event */
  if ((state == 1) && ((xTaskGetTickCount() - start_time) >= MSEC2TICK(300)))
    input_event = TO;
  else if (NDS_SWITCH() & KEY_A)
    input_event = SW_ON;
  else
    input_event = SW_OFF;

/* Step 1: Do Action */
  switch (Exam_SM_B[state][input_event].action) {
    case TS :
      start_time = xTaskGetTickCount();
      break;
    case LED1 :
      writeb_virtual_io(BARLED1, 0xFF);
      writeb_virtual_io(BARLED2, 0);
      break;
    case LED2 :
      writeb_virtual_io(BARLED1, 0);
      writeb_virtual_io(BARLED2, 0xFF);
      break;
    case NOP :      // No operation
    default :       // Error !, No nothing
      break;
  }
/* Step 2: Set Next State */
  state = Exam_SM_B[state][input_event].next_state;

  if (NDS_SWITCH() & KEY_START)
    break;
  vTaskDelay(MSEC2TICK(50));
}
```

Exp_3_Sample_B()와 같이, 우리 상태표는 간단하고 동작이 몇 가지 안되기 때문에, 세 개의 동작과, Nop(No operation)을 enum으로 정의한 뒤, 각각을 Step1에서 switch() 문를 이용하여 구동하였으나, 실제 상태도 많고 동작도 많은 경우에는, Step 1을 위해서 긴 switch() 문이 만들어져 소스가 읽기 어려워지고, 관리도 쉽지 않다. 따라서 상태도 많고, 동작이 매우 많은 경우에는 각 동작을 함수로 구현하고, 각 동작 함수를 상태표에 함수 포인터로 등록한 뒤, 그것을 호출하는 방법을 사용한다.

또, TO (Timeout) 이벤트를 if 문에서 상태와 함께 비교를 하였지만, 이 부분 역시 상태와

무관하게 타이머 ID를 이용하여 일반적인 형태로 구현할 수 있다.

실제로 상태가 많고, 여러 개의 타이머가 사용되는 환경에서는 Exp_3_Sample_C()와 같은 코드를 사용한다. (소스는 일부 생략한 것으로 전체 소스는 Exp_Sample.c 참조)

```
struct state_machine_c {
  int check_timer;
  int next_state[NUM_INPUT];
  void (* action[NUM_INPUT])(void *p);
};

static
void
f_led1(void *p)
{
  writeb_virtual_io(BARLED1, 0xFF);
  writeb_virtual_io(BARLED2, 0);
}
...

struct state_machine_c Exam_SM_C[NUM_STATE] = {
    { 0, { 1, 0, 0 }, { f_ts, NULL,   NULL } },    /* State 0 */
    { 1, { 1, 0, 2 }, { NULL, f_led1, NULL } },    /* State 1 */
    { 0, { 2, 0, 2 }, { NULL, f_led2, NULL } }     /* State 2 */
};
...

while (1) {

/* Step 0: Generate Input Event */
  if (Exam_SM_C[state].check_timer) {
    if ((xTaskGetTickCount() - start_time_c) >= MSEC2TICK(300)) {
      input_event = TO;
      goto do_action;// Input happens
    }
  }
  if (NDS_SWITCH() & KEY_A)
    input_event = SW_ON;
  else
    input_event = SW_OFF;

/* Step 1: Do Action */
do_action:
  if (Exam_SM_C[state].action[input_event]);
    Exam_SM_C[state].action[input_event](NULL);

/* Step 2: Set Next State */
  state = Exam_SM_B[state][input_event].next_state;
...
```

4. 응용 실험 (과제)

이 프로그램에서는 스위치 A (KEY_A)를 짧게 한번 Click을 하면 BARLED2 가장 왼쪽부터 오른쪽으로 차례로 LED가 하나씩 점점 더 켜지며, 짧게 두 번 Click (보통 더블 클릭)을 하면 LED가 거꾸로 LED가 하나씩 차례로 꺼진다. 그리고 길게 누른 뒤 바로 짧게 누르면 (더블 클릭의 앞을 길게 누르면) BARLED1, BARLED2의 모든 LED가 켜지고, 길게 두 번 누르면 (더블 클릭의 두 번 다 길게 누르면) 모든 LED가 꺼져 초기 상태로 돌아간다. 또 길게 한 번 누르면 왼쪽 의 BARLED1 전체만, 짧게 누른 뒤 바로 길게 누르면 BARLED1의 왼쪽부터 6개의 LED만 켜진다. (아래 표 참조). (BARLED1, BARLED2의 모든 LED가 모두 켜졌을 때, Short Click, 또는 모든 LED가 꺼졌을 때 Short-Short 클릭 때는 상태 변화가 없다.)

입력	BARLED 상태
(초기상태)	모두 꺼짐
Short Click	누를 때마다 왼쪽부터 하나씩 차례로 켜짐
Long Click	BARLED1만 전체가 켜짐
Short-Short Double Click	Short Click과 반대로 차례로 꺼짐
Short-Long Double Click	BARLED1의 왼쪽부터 6개 켜짐
Long-Short Double Click	BARLED1, BARLED2 모두 켜짐
Long-Long Double Click	모든 LED 꺼지고, 초기 상태가 됨

위에서 Short Click은 200msec이하 시간 동안 누른 것을, Long Click은 그 이상 길이로 누른 것을, 더블 클릭의 사이 간격은 200msec 이하로 정의한다.

위 표의 동작을 구현하기 위한 상태표 또는 상태 다이어그램을 결과 보고서에 “반드시” 그리고, 그 상태표(상태 다이어그램)대로 소프트웨어를 구현한다. (실험 시작 때, 상태표와 그 구현을 과제로 해왔는지 검사 함)

프로그램은 “반드시” 예제 구현 방법에 설명된 함수 포인터를 이용하는 Exp_3_Sample_C() 방법으로 구현한다.

5. 보고서 제출

보고서는 Sourseforge.net의 Template을 참고하여 실험 검사 완료 후 제출

6-1. 소스 및 샘플 프로그램 (게시판 이용)

A. Template 소스는 자료실의 Simple-IO-3.zip에 있음
B. 미리 만들어 놓은 Binary는 자료실의 Simple-IO-3.nds 임

6-2. 소스 및 샘플 프로그램 (Github 이용)

소스는 Github의 simple-io-3 소스를 check-out하여 과제를 수행한다.

https://github.com/hl1itj/nds-ide/tree/microprocessor/lab/simple-io-3

에서, Checkout하여 과제를 수행하고
(이 문서와 미리 build한 binary Simple-io-3.nds는 doc directory에 있음)

https://github.com/hl1itj/nds-ide/tree/microprocessor/학번/simple-io-3

에서 각자 개발을 진행한다.